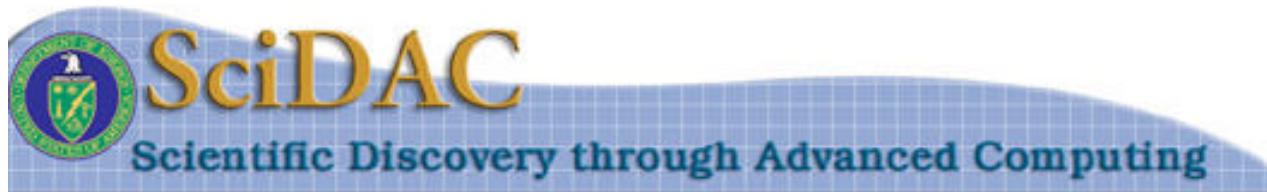


Session II: SvPablo **10:30 AM – 12:30 PM**

Ying Zhang
zhang8@cs.uiuc.edu

University of Illinois at Urbana-Champaign



Outline

✍ SvPablo Introduction

- ✍ Goal and Overview
- ✍ Architecture

✍ Obtaining and Installing SvPablo

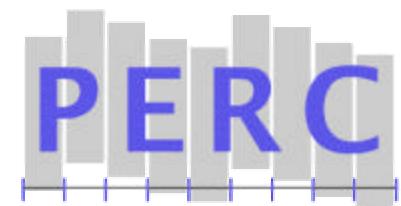
✍ Using SvPablo

- ✍ Instrument source code
- ✍ Compile and run the instrumented code
- ✍ Browsing performance data

✍ Experiments with POP

✍ SvPablo Current status

✍ Hands-on Session Overview



SvPablo Goals

- ✍ **Automate generation of instrumented executable code**
- ✍ **Correlate performance data to lines and fragments in the source code**
- ✍ **Support scalability analysis and prediction**
- ✍ **Support multiple languages and portability across HPC systems**



SvPablo Overview

☞ Three major components:

☞ Source code instrumentation

- ☞ Loops and function calls, for C, Fortran77, Fortran 90

☞ Performance data capture

- ☞ Software statistics and hardware counter data

☞ Performance data presentation and analysis

- ☞ Line and procedure level data browsing with the source code

☞ Supported platforms

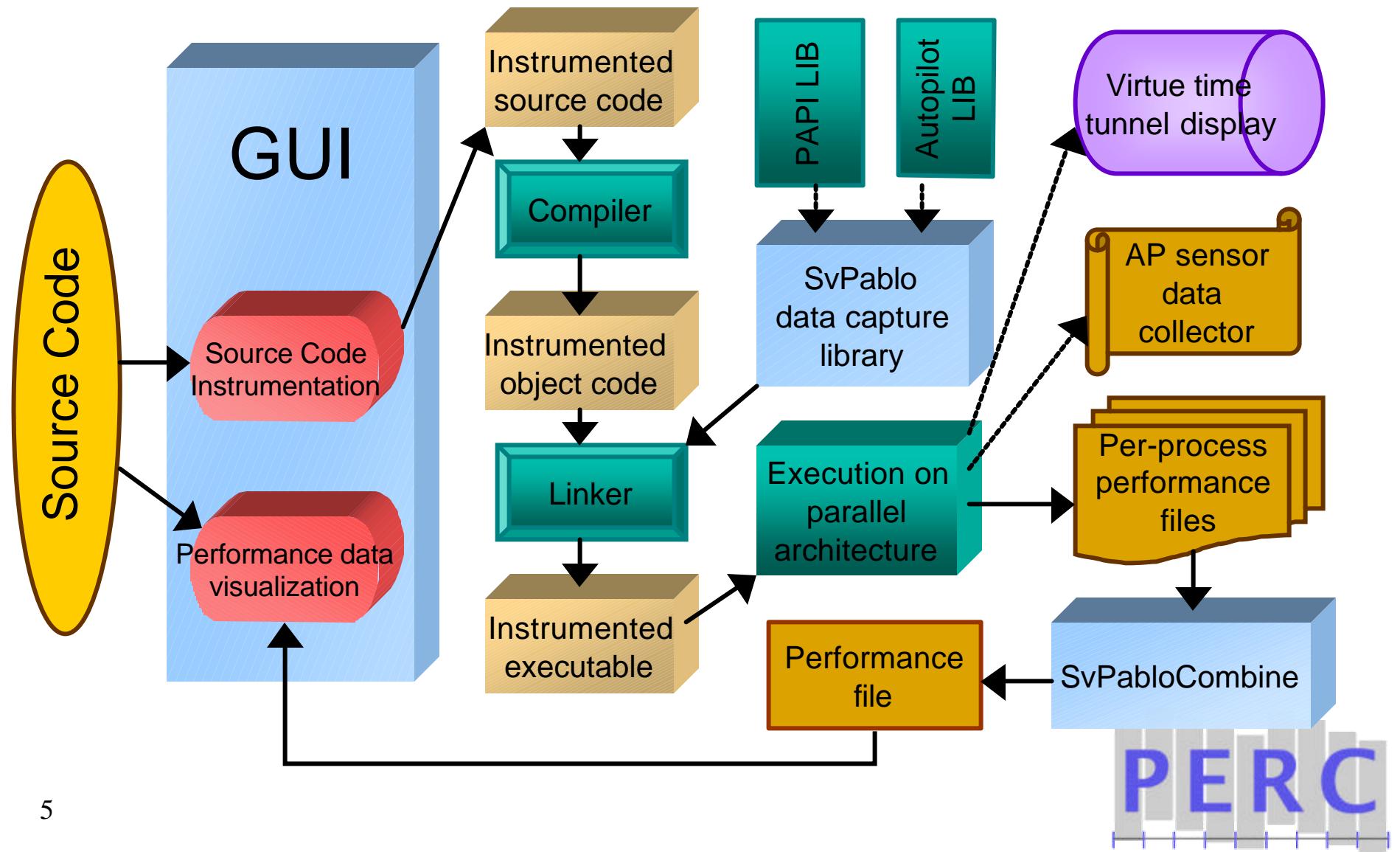
- ☞ IBM SP, HP/Compaq Alpha, SGI Origin, Sun Solaris, NEC SX6 (limited tests)

- ☞ Linux (IA-32, IA-64, PlayStation-2)

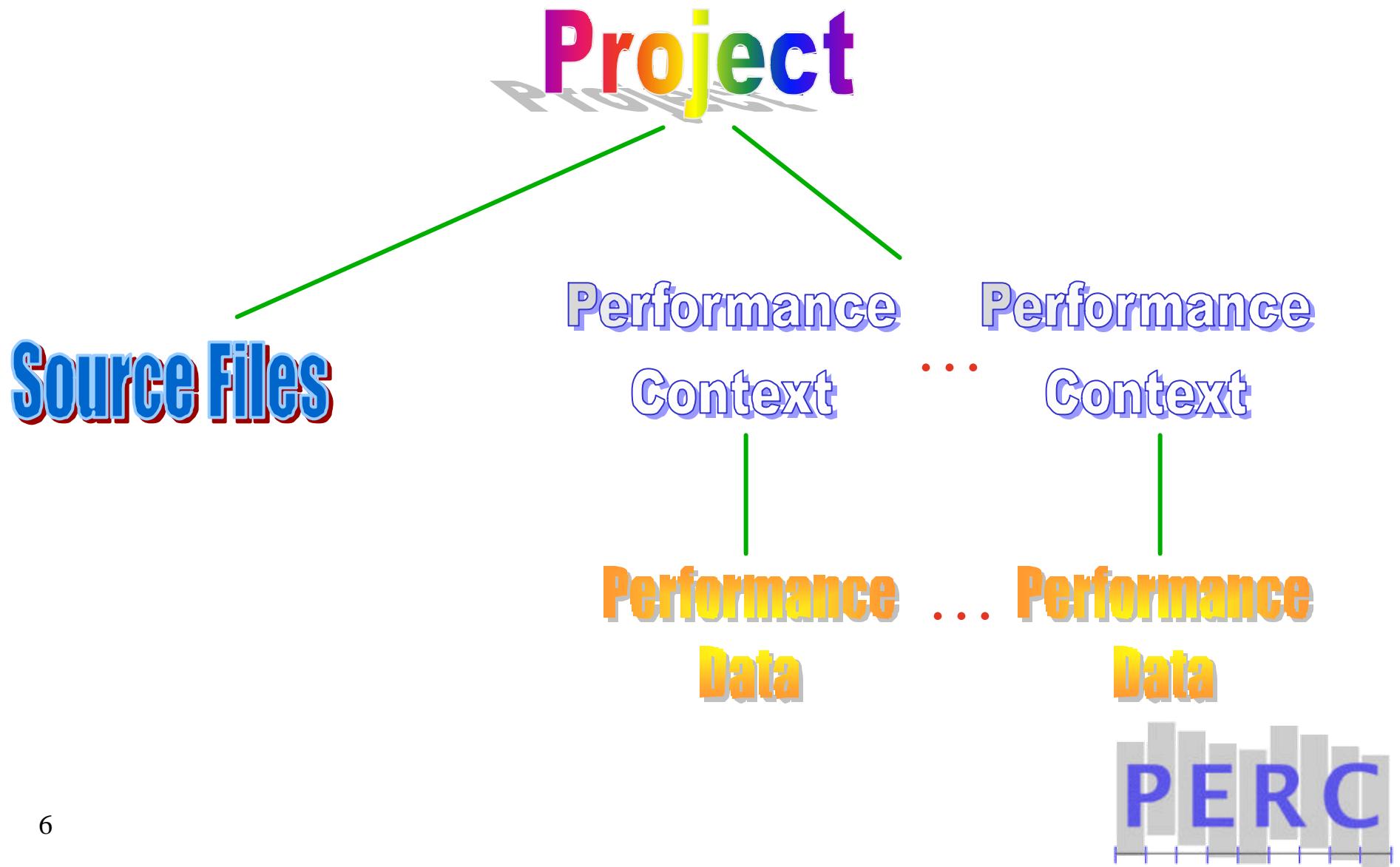
☞ Latest release version 5.2



SvPablo Components



SvPablo File Structure



SvPablo Installation

☛ Download SvPablo from SvPablo website

<http://www-pablo.cs.uiuc.edu/Software/SvPablo/svPablo.htm>

☛ Software Requirements

- ☛ Pablo *SDDF* library
- ☛ MPI library for parallel programs using MPI
- ☛ Motif library (*Lesstif* is used for Linux operating systems)
- ☛ *PAPI* library if hardware counter performance data is desired

☛ Hardware Requirements

- ☛ Supported on major parallel platforms.

☛ Documentation

- ☛ *SvPablo User's Guide* (available on SvPablo website)

☛ SvPablo Release Packages

- ☛ Complete source release including documents, examples
- ☛ Ready-to-install binary releases for various platforms



SvPablo Installation (Continue...)

Build SvPablo From Source Release

Modify the file **Makefile.defines**

 Specify SvPablo installation path

 Specify installation paths for SDDF, MPI, PAPI (optional)

Type command

% gmake install

Build SvPablo From Binary Release

Modify the file **fixPaths**

 Specify installation path for SvPablo, MPI

Run the script **fixPaths**

% ./fixpaths



SvPablo Usage Introduction



Four Steps

1. Create a new SvPablo project and performance context
2. Instrument source code
3. Compile and run the instrumented code
4. Browse performance data in SvPablo GUI



Repeat

- >Create a new performance context
- Repeat from Step 2 or Step 3

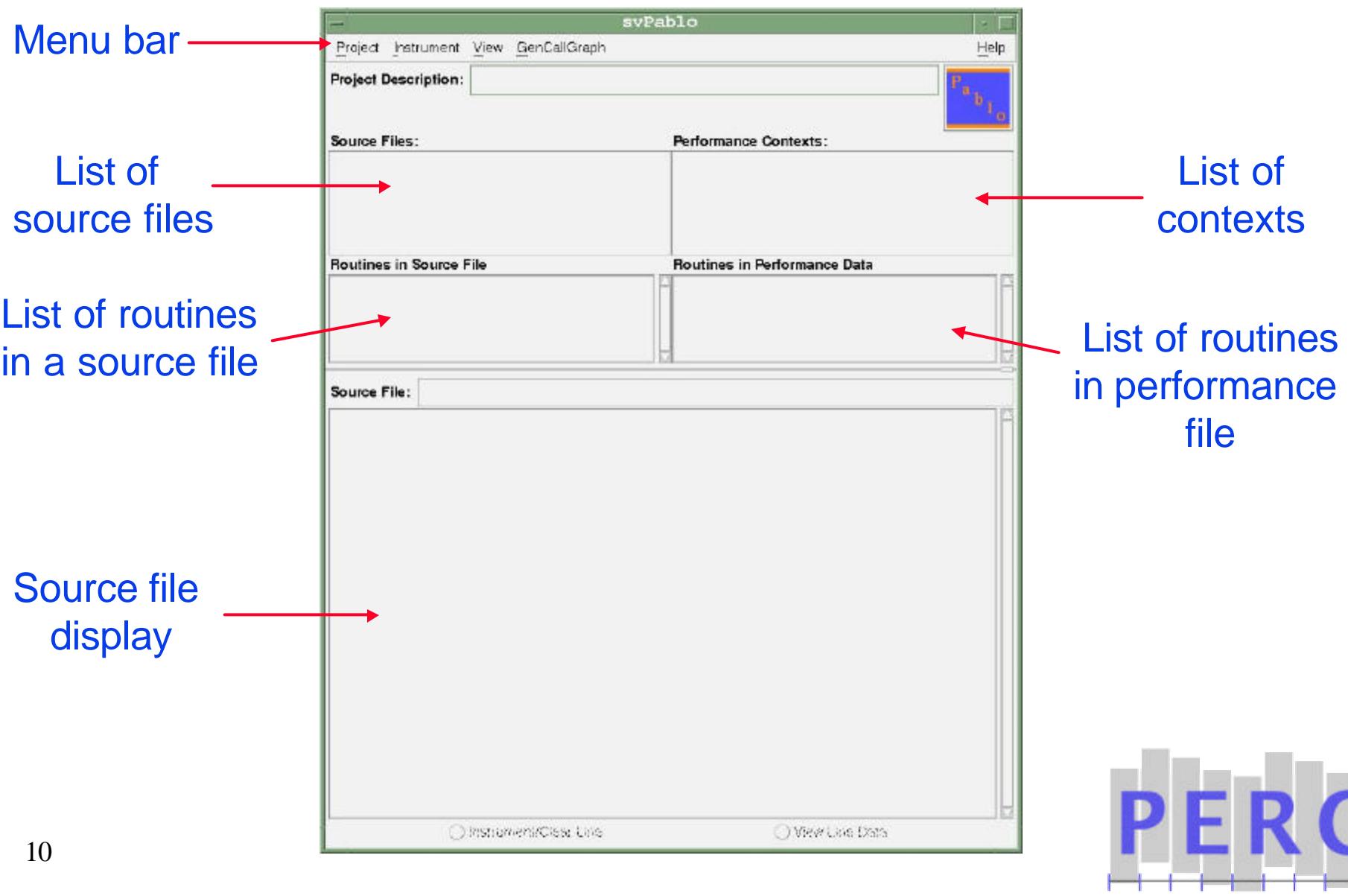


Running SvPablo

- In `$(INSTALL)/bin` directory, run script
`% ./runSvPablo`



SvPablo Main Window

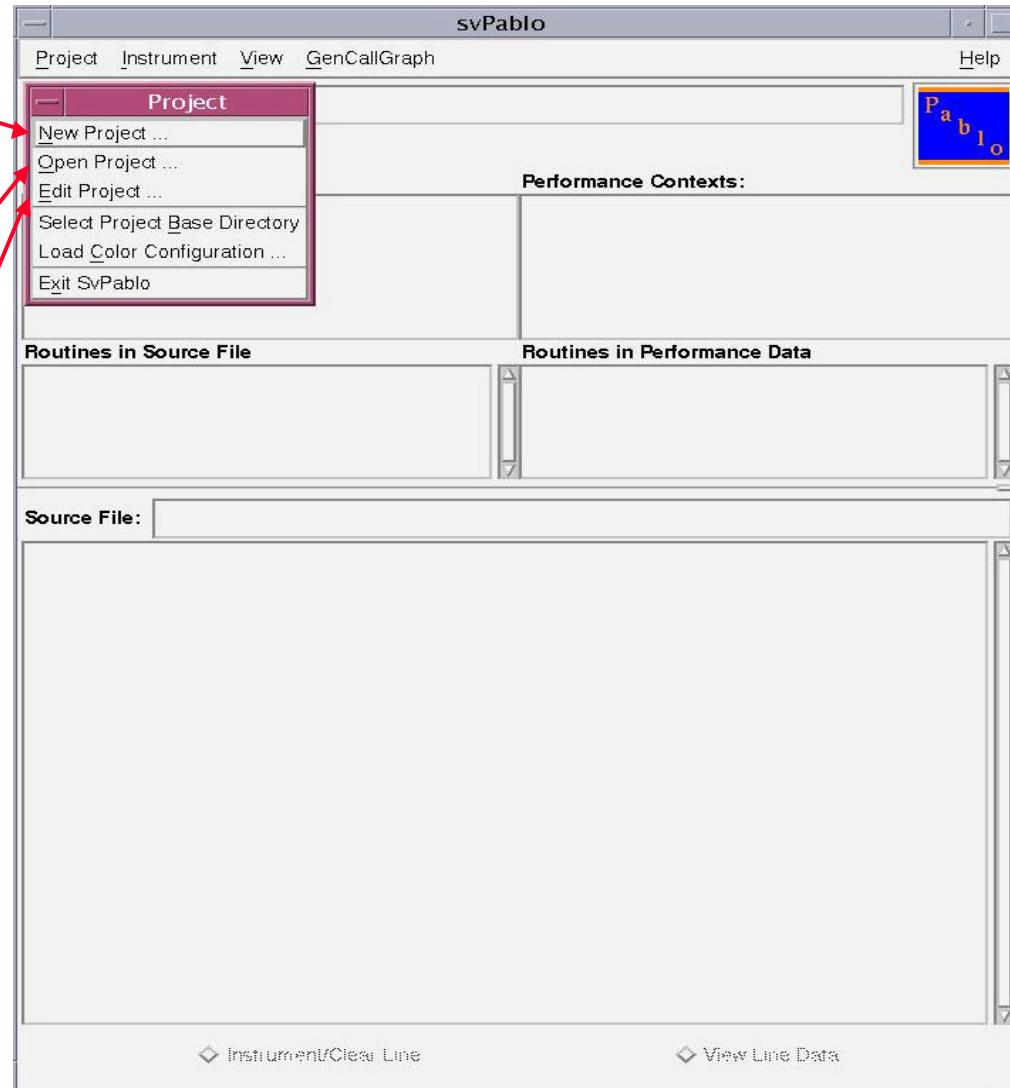


Create SvPablo Project

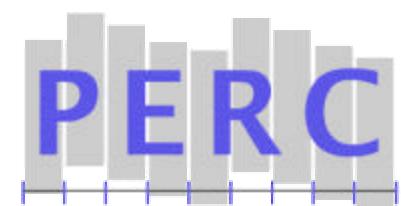
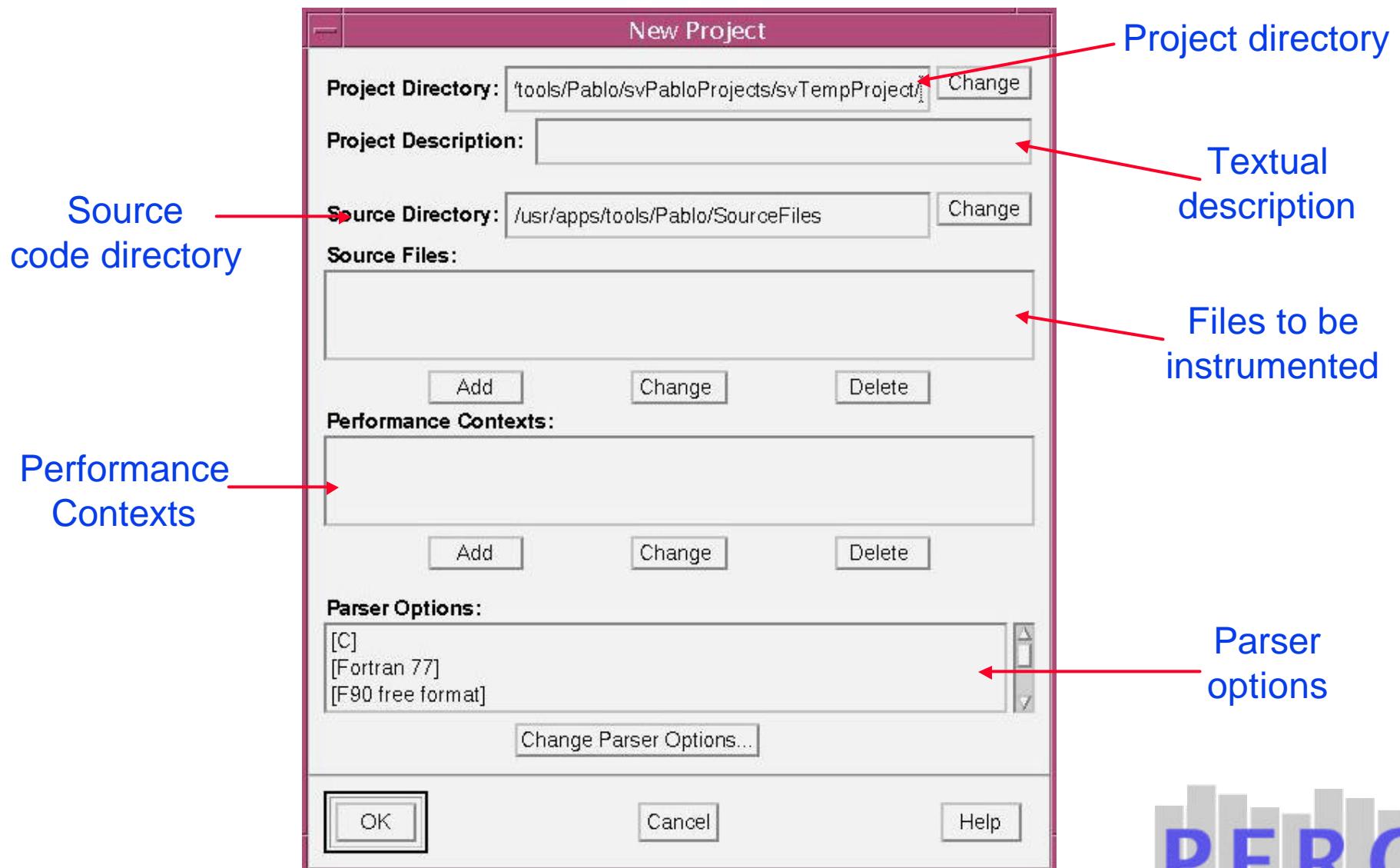
Create New
project

Open existing
project

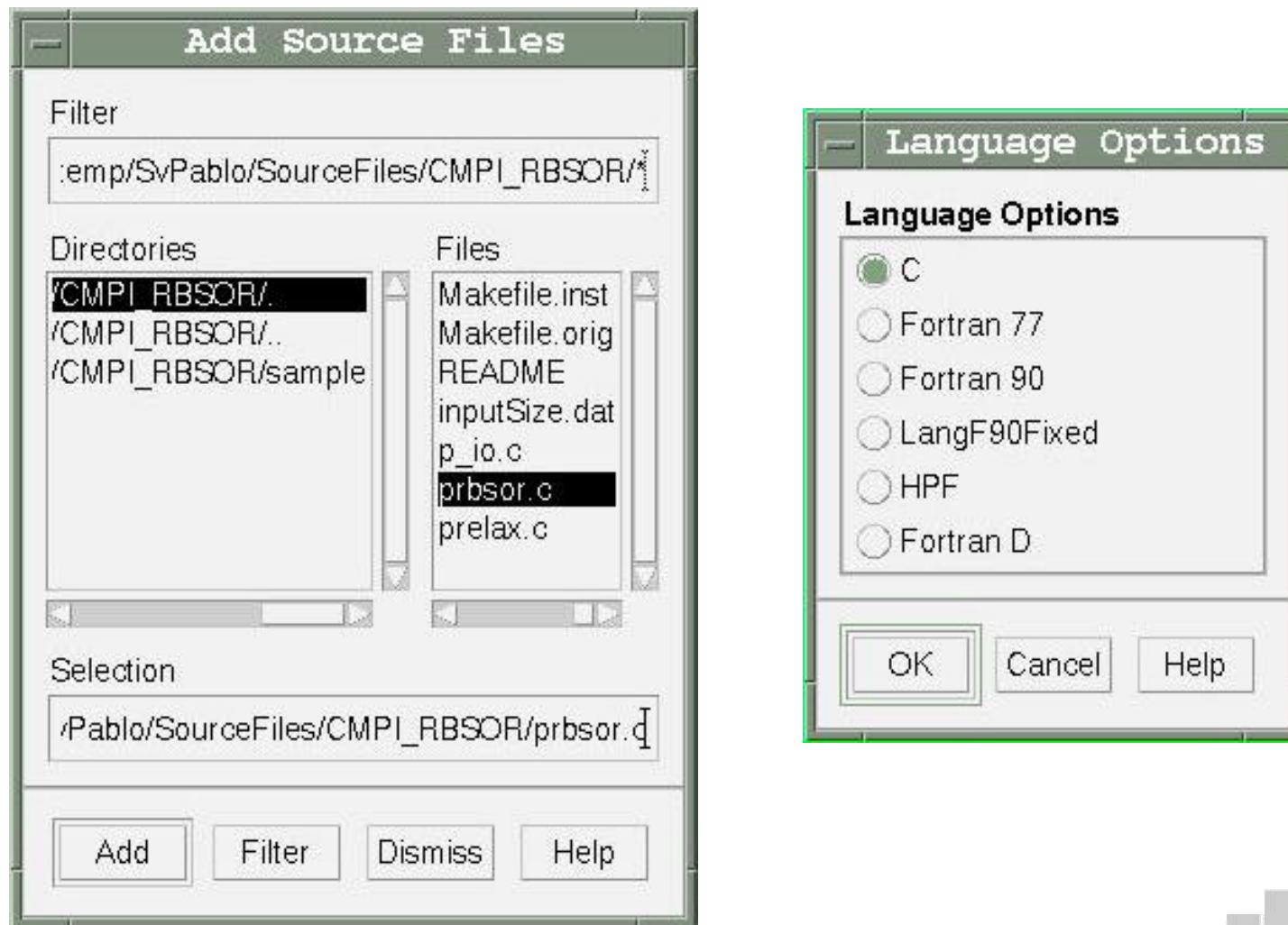
Edit
existing project



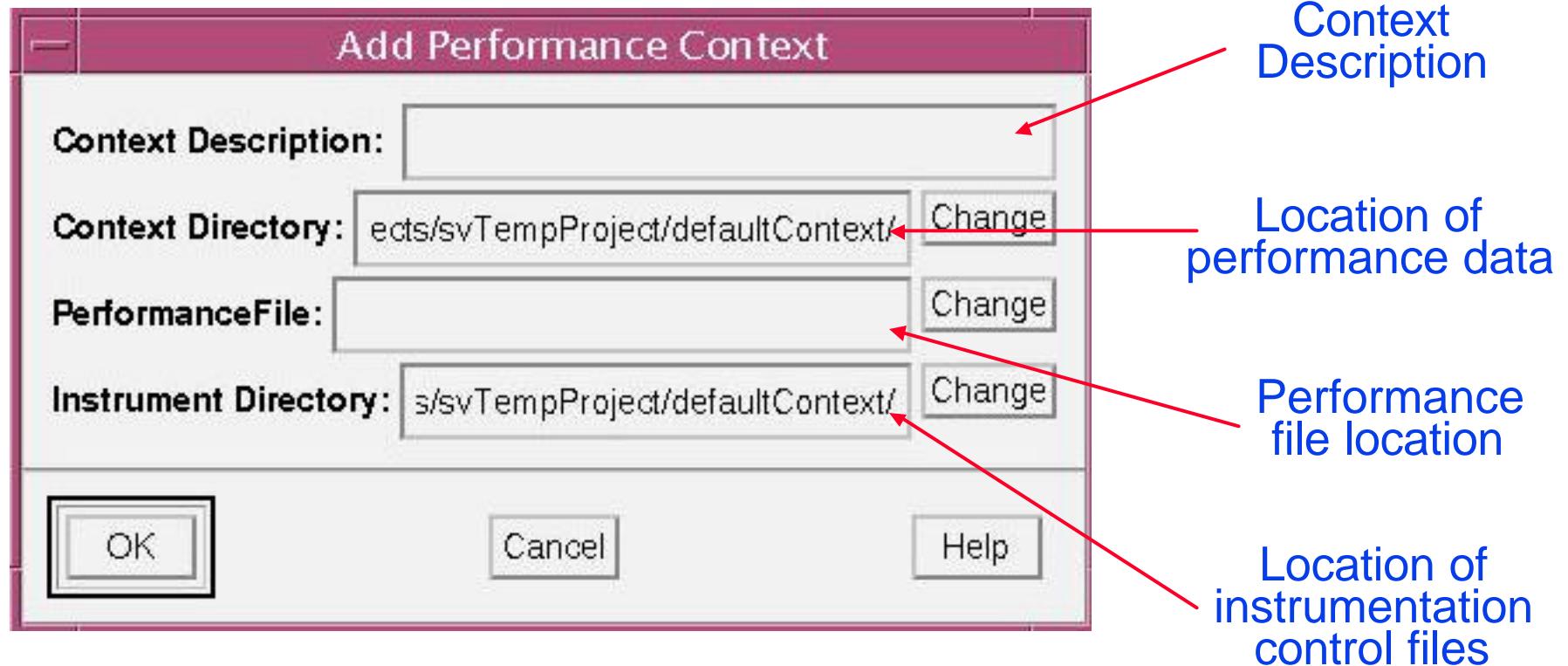
Project Creation Box



Adding Source File Box



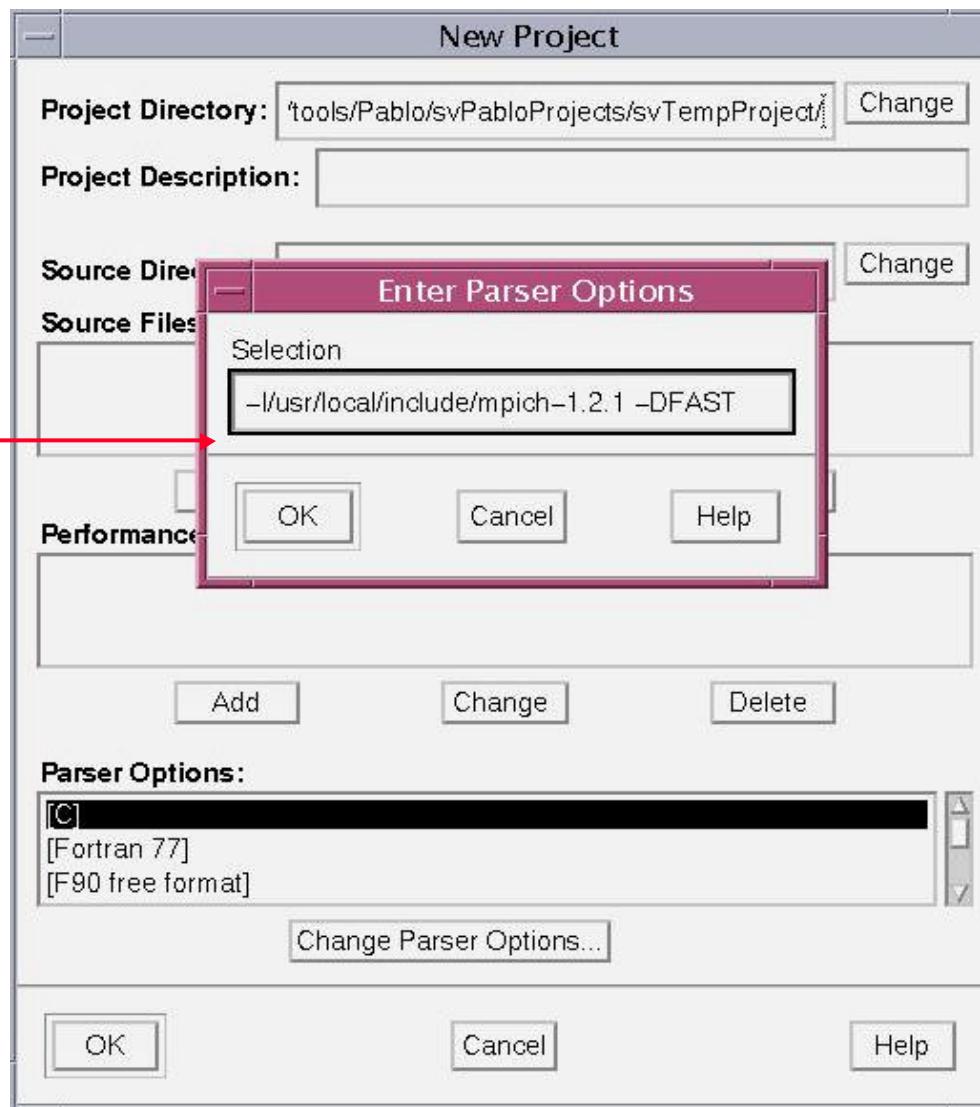
Performance Context Editing Box



☞ Typically “Context Directory” = “Instrument Directory”

Parser Option Editing Box

Options necessary
to parse
source code



Source Code Instrumentation

✍ Instrumentation Constructs

✍ Function calls

✍ Outer loops

✍ Two Options

✍ Interactive Instrumentation

✍ Via SvPablo GUI

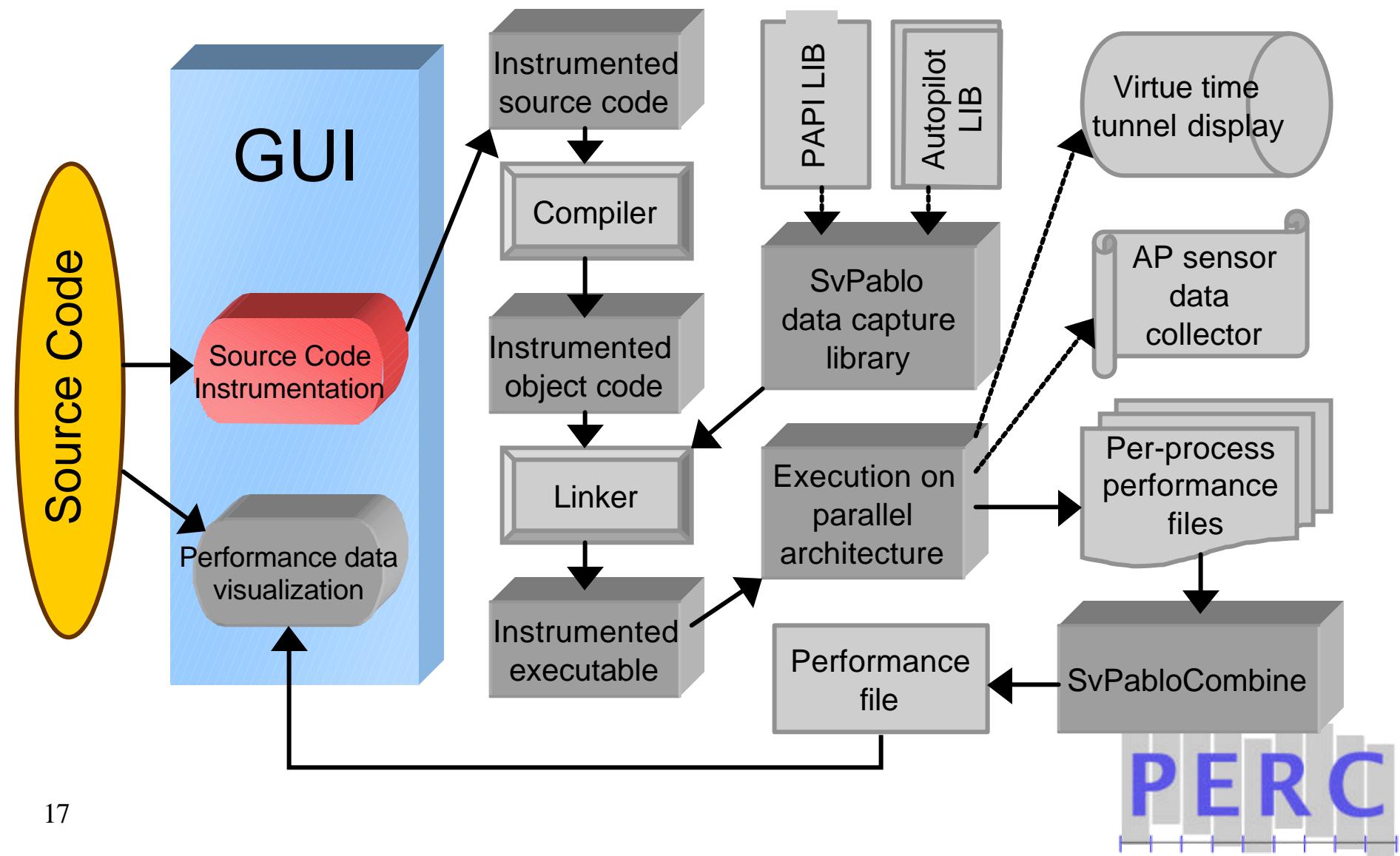
✍ Automatic Intrumentation

✍ Via Standalone Parser – command line

% SvPabloParser [options] sourcefiles...



Source Code Instrumentation via GUI



Instrument Line by Line

SvPablo

Project Instrument View GenCallGraph Help

Project Description: Red Black SOR in C using MPI

Source Files: prbsor.c prlax.c p_j0.o

Performance Contexts: No Instrumentation Hockney 2 processors

Routines in Source File: main MPI_Comm_size MPI_Comm_rank MPI_Get_processor_name printf

Routines in Performance Data

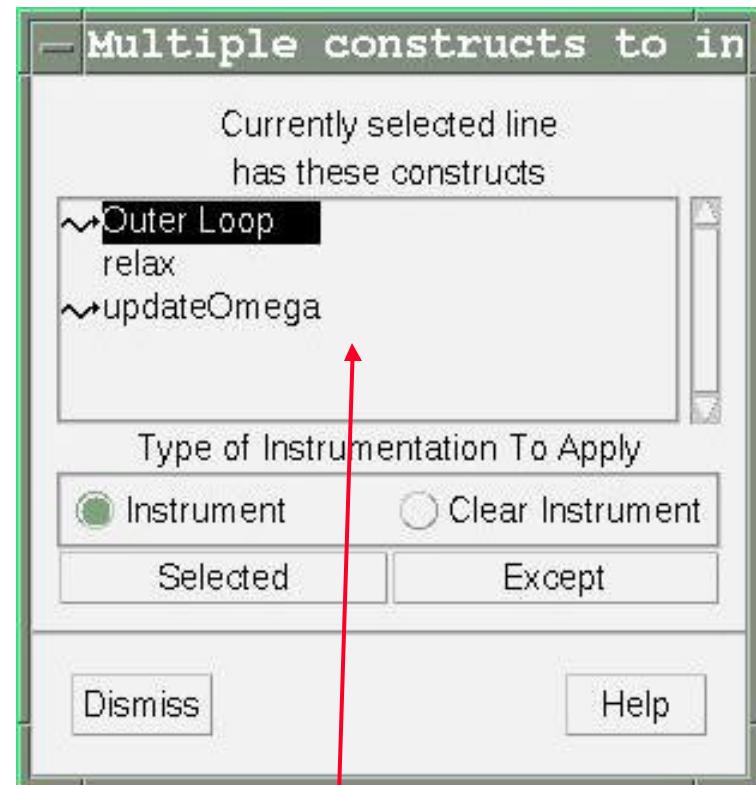
Source File: /usr/common/homes/yingz/temp/SvPablo/SourceFiles/CMPI_RB9OR/prbsor.c

```
~> for ( i=0; i <= myend + 1; i++ )
    {
        for ( k=0; k< n; k++ )
        {
            u[n * i + k] = 0.0;
            f[n * i + k] = rbs;
        }
    }

/*
 * The following line has statements grouped together
 * to test some functionalities of the SvPablo GUI.
 */

~> for (i=1; i<=it; i++) | relax(h,&omega,f,u,&mynorm); updateOmega(&omega);
    > MPI_Sendrecv( su[myend:n], n, MPI_DOUBLE, bottom, (myid+1)*blocksize,
        n, n, MPI_DOUBLE, top, myid * blocksize,
        MPI_COMM_WORLD, &status );
    > MPI_Sendrecv( su[n], n, MPI_DOUBLE, top, myid * blocksize + 1,
```

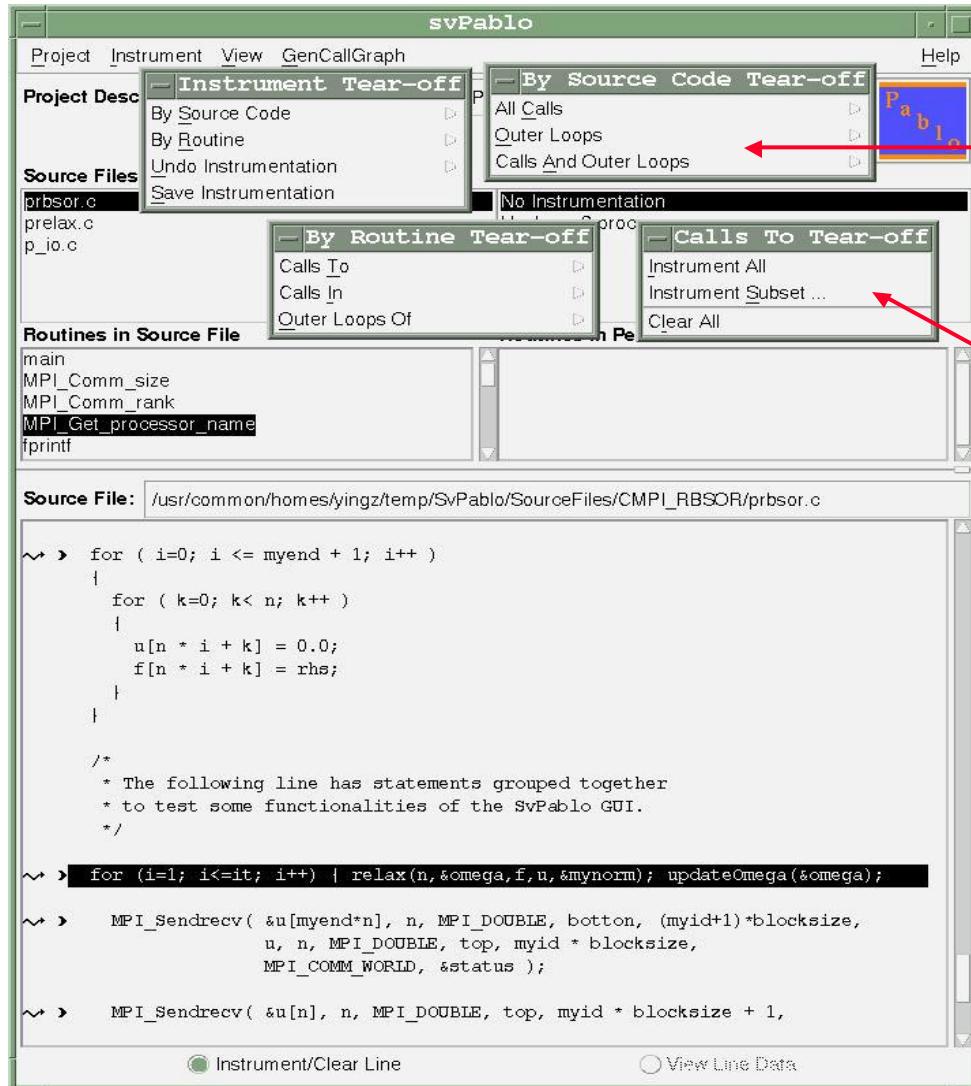
Instrument/Clear Line View Line Data



Multiple events in one line



Instrument by Groups



Instrument by
Source code

Instrument by
routine

Automatic Instrumentation

✍ Standalone Parser

✍ Usage

✍ % SvPabloParser [options] [cpp options] filenames

✍ Options

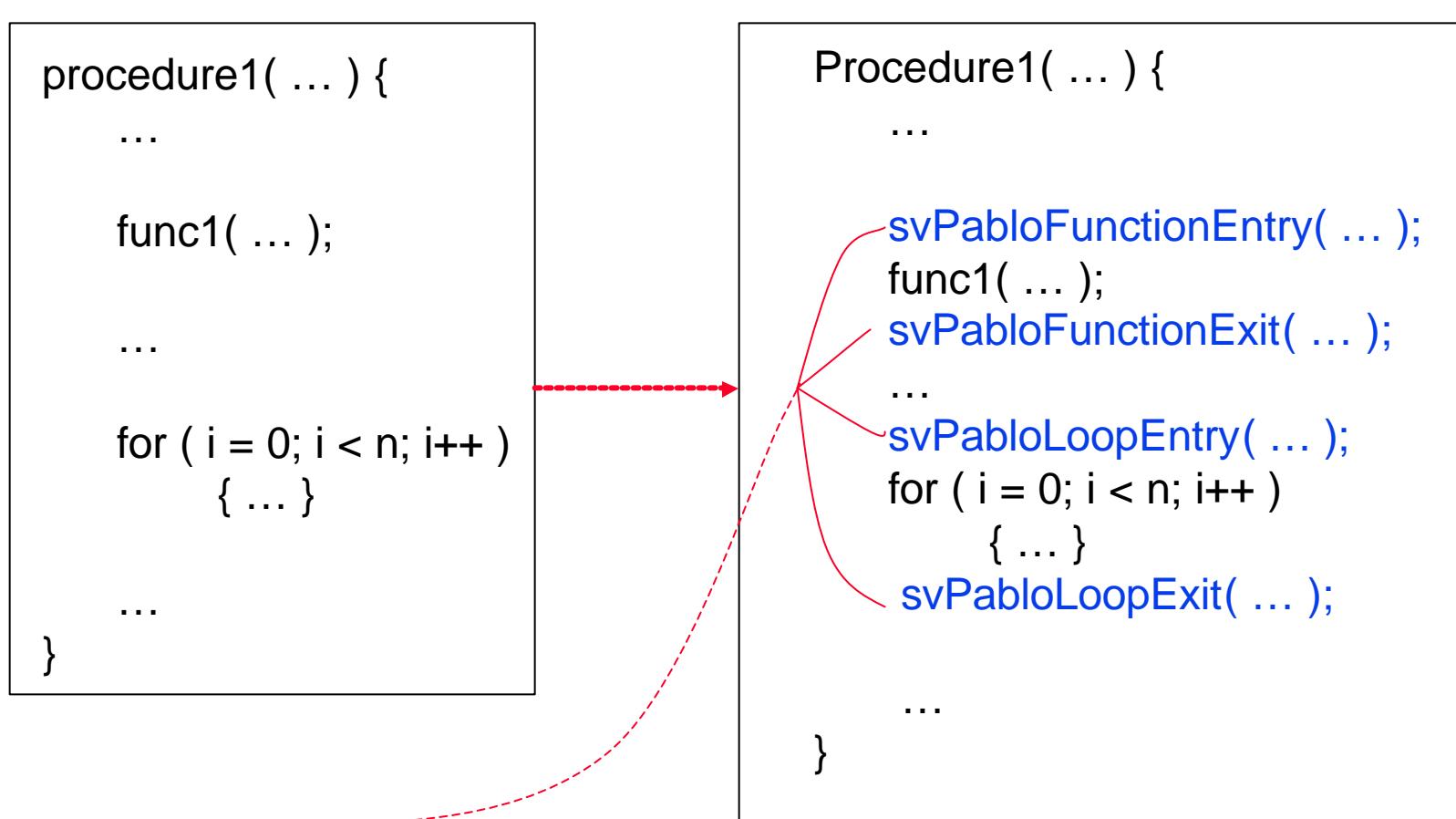
- ✍ -a: instrument all function calls and outer loops
- ✍ -s: instrument all function calls
- ✍ -l: instrument all loops
- ✍ -o: instrument OpenMP constructs
- ✍ -g: generate call graphs
- ✍ -c<context name>: specify the performance context name, default is “default”

✍ CPP options

- ✍ -I: specify the header files
- ✍ -D<macro>: specify to instrument the code segment between #ifdef<macro> and #endif
- ✍ -F<cpp filename>: specify the file which contains user specified CPP options

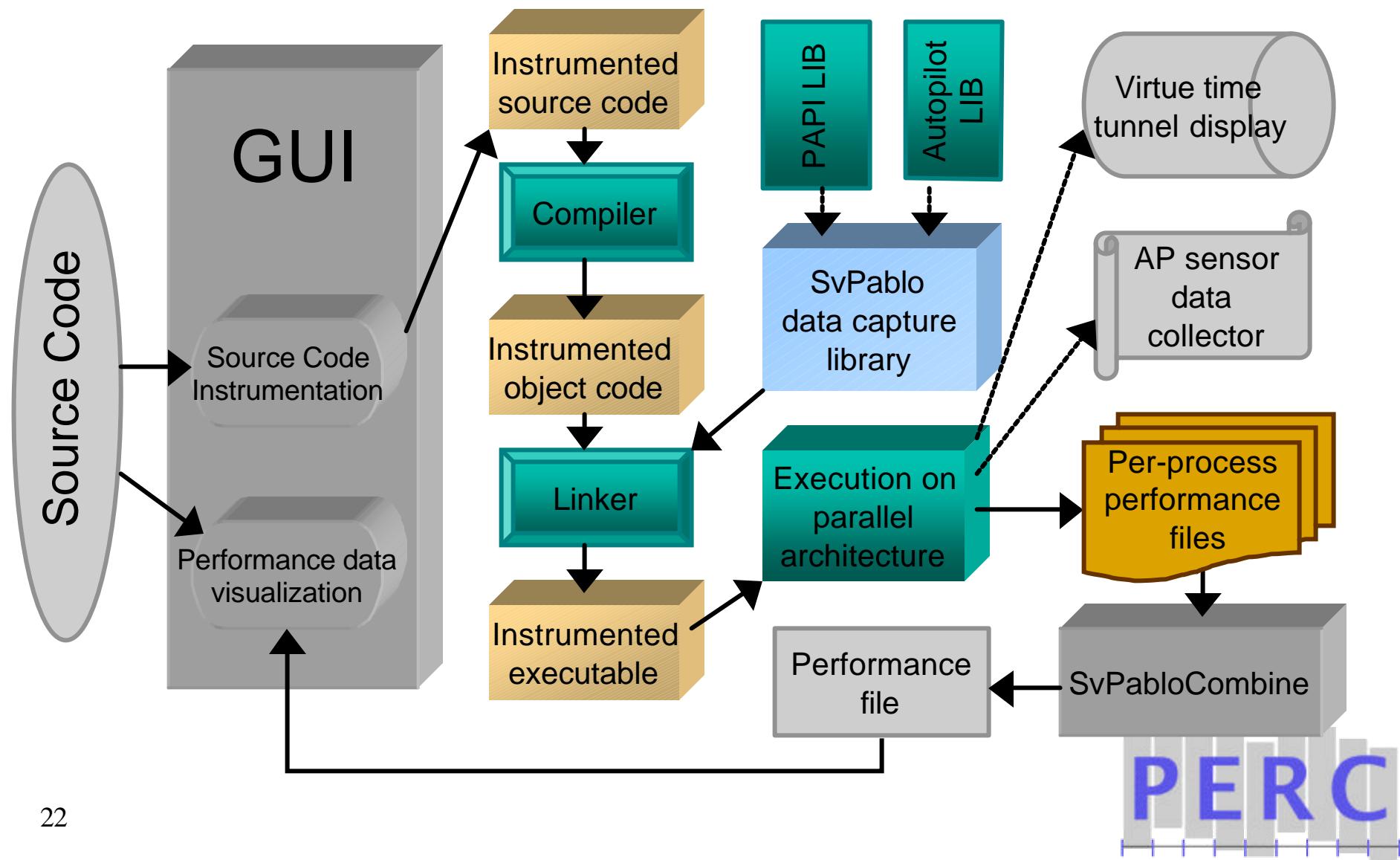


Inside Instrumentation



Defined in SvPablo library

Compiling and Running Instrumented Code



Compiling Instrumented Code

✍ Makefile modification

✍ Add InstrumentationInit.c into compile and linking

- ✍ Initialize SvPablo performance data collection
- ✍ Generated by SvPablo instrumentation

✍ Replace source file names with saved instrumented file names

e.g. pop.F -> pop.\$(contextname).inst.F

✍ Link with SvPablo Library

- ✍ libsv_DCL.a for parallel applications
- ✍ libsv_SEQ.a for sequential applications

✍ Link with hardware counter library

- ✍ PAPI on Linux, IBM SP, HP/Compaq Alpha
- ✍ On SGI Origin, no need to link explicitly



Running Instrumented Code

- ✍ Choosing Hardware Counter Events

- ✍ Specify desired hardware counter metrics in file
svPabloHWevents

```
1 0 native# Cycles# Cycles#
0 0 PAPI_FP_INS# FP Instructions# Floating Point Instructions#
0 0 PAPI_L1_DCM# L1 D-Cache Misses# Level 1 Data Cache Misses#
```

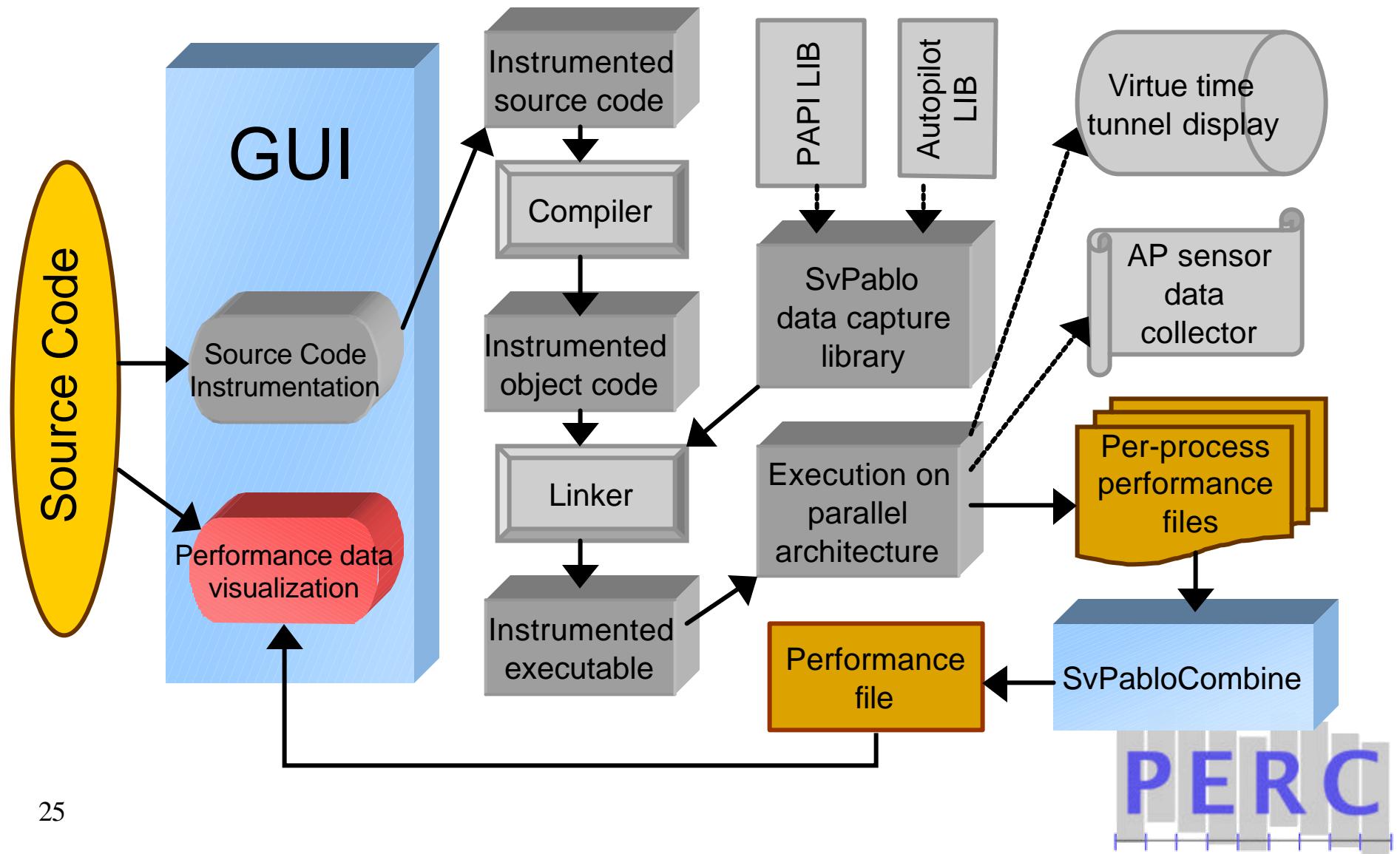
- ✍ Use the default events defined in SvPablo library

- ✍ Running the Instrumented Code

- ✍ Run the code in the same way as to run the original code



Performance Data Visualization



Merging Per-task Performance Data

- ☞ One performance file is generated for each task

c_sddf0000.ascii, c_sddf0001.ascii, ...

- ☞ Combining per-task files into one performance file

☞ Use *SvPabloCombine* utility

- ☞ Control file *projectHistory*, generated by instrumentation

- ☞ Each instrumentation generates one *projectHistory* file

- ☞ Syntax

```
% SvPabloCombine -o outputFile [-h projHistFile] c_SDDF*.asci
```

- ☞ Only the performance file generated by *SvPabloCombine* can be browsed in SvPablo GUI



SvPablo Performance Data

- ✍ Basic Statistics at Line and Procedure level

✍ Counts

✍ Durations: inclusive and exclusive

- ✍ Hardware Counter Performance Data at Line Level

✍ Any metrics provided by PAPI or SGI hardware counter interface

✍ Synthesized metrics

✍ MFLOPS

✍ % branches mispredicted

- ✍ Detailed performance data is available for each task

- ✍ Line Level performance data correlates to the original source code

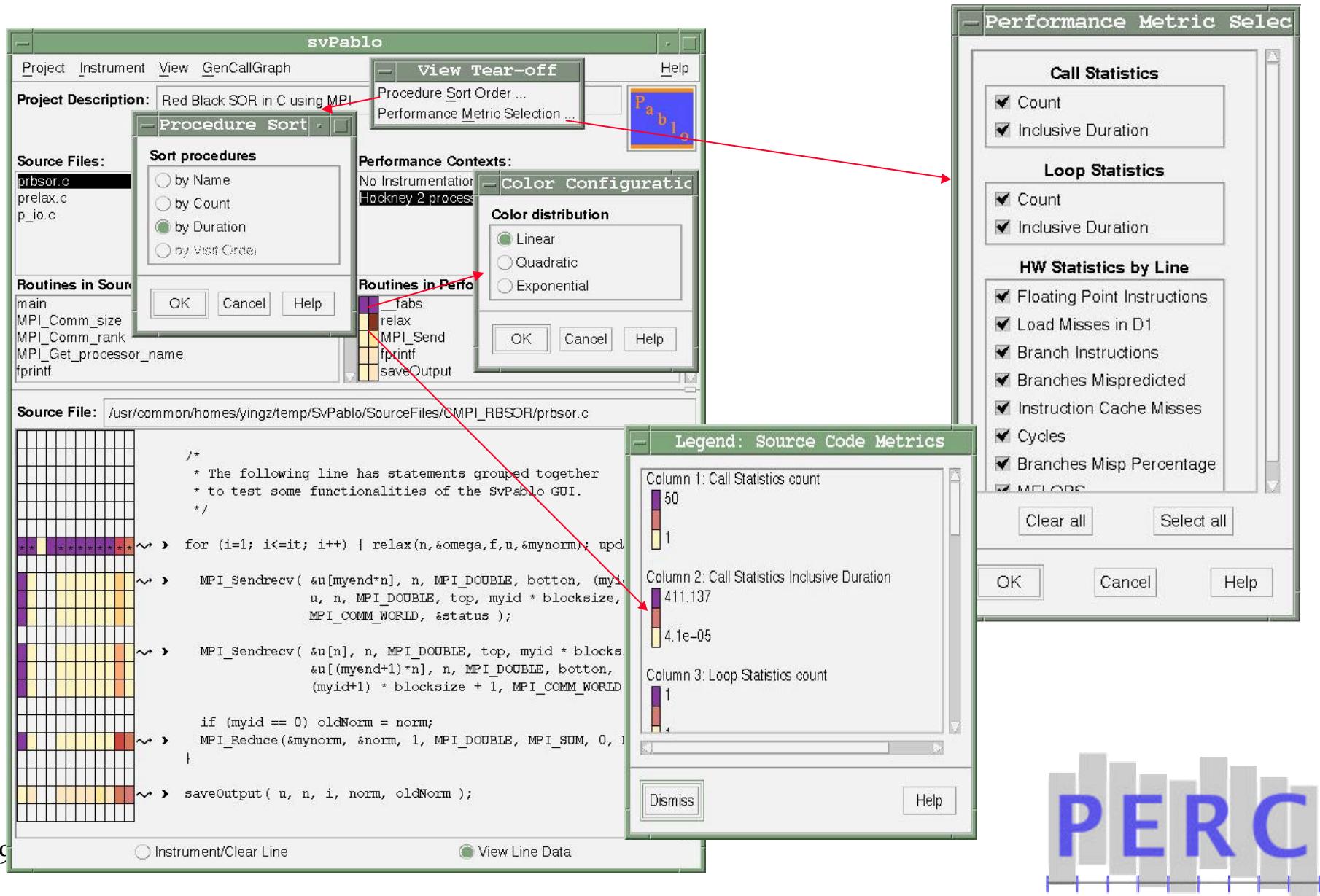


Performance Data Browsing in GUI

- ☞ **Color encoded data display**
 - ☞ Configurable by the user
 - ☞ Easy to detect bottlenecks
- ☞ **Statistical performance data**
 - ☞ Mean and standard deviation values across tasks
 - ☞ Maximum and minimum values and corresponding task number
- ☞ **Selectable performance metrics display**

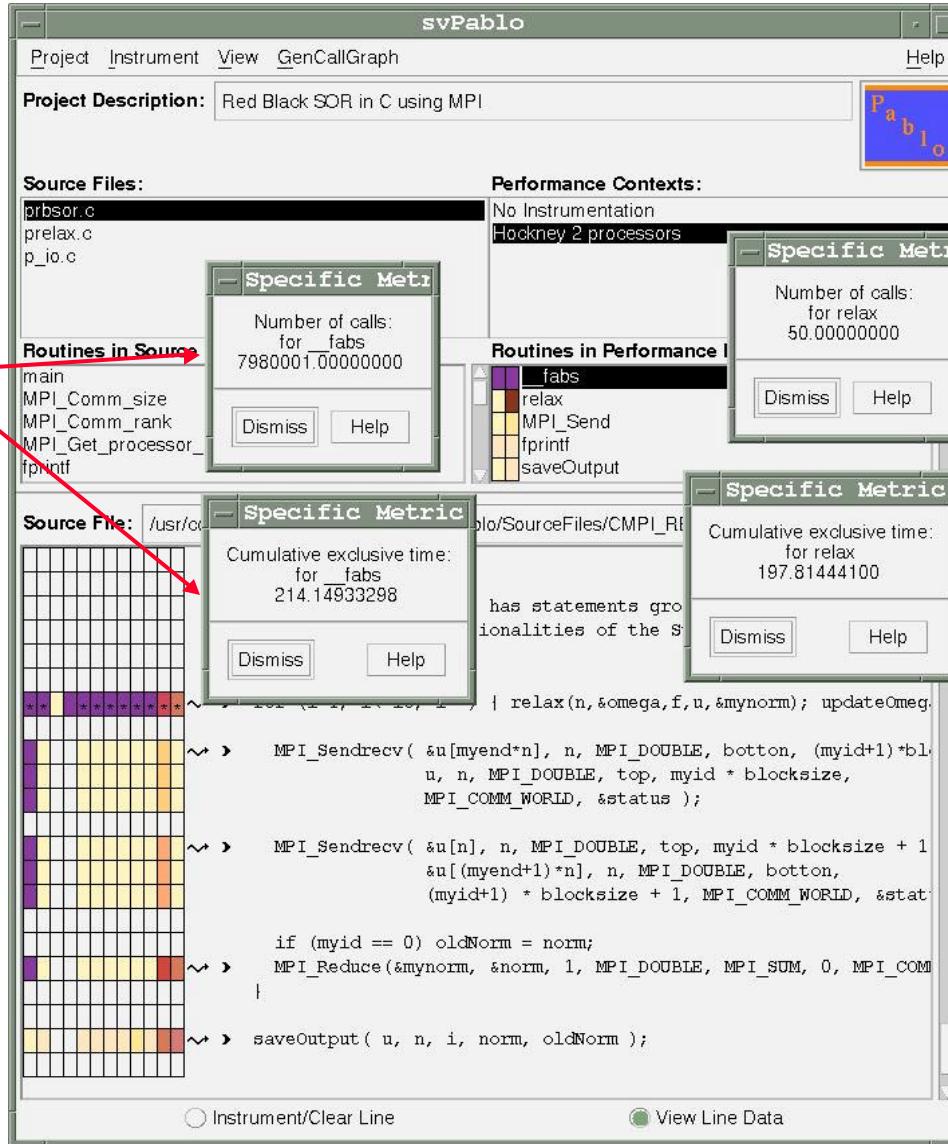


Basic GUI Display Choice



Procedure Level Performance Data

Counts and duration
For routine `_fabs()`



Counts and duration
For routine `relax()`



Procedure Level Performance Data

Performance Data: Procedure Statistics

Tasks: 0 .. 3

File Name: prelax.c Routine Name: relax()

Line Number: 114

Source Code Fragment:

```
/*
 * File:    prelax.c
 * Description: This MPI program performs a red black SOR using Chebyshev
```

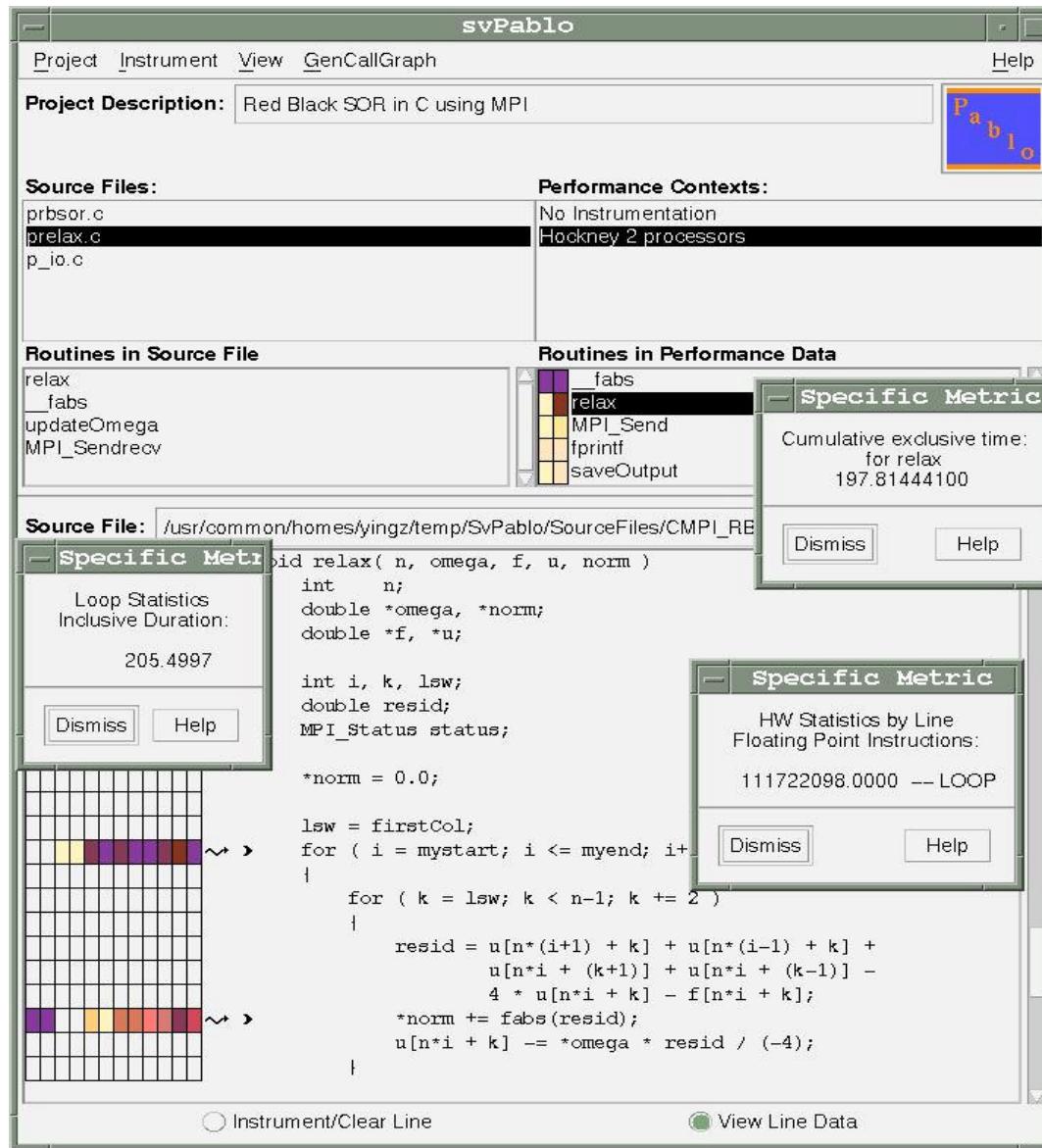
Overall Procedure Performance Data

Field Name	Mean	Max	Min	Std Dev		
	Value	Task	Value	Task		
Count	50.000000	50.000000	0	50.000000	0	0.00000000
Exclusive Duration	195.743635	197.814441	0	193.448951	3	1.72332407
Inclusive Duration	408.890388	411.137097	1	406.555593	3	1.76721129

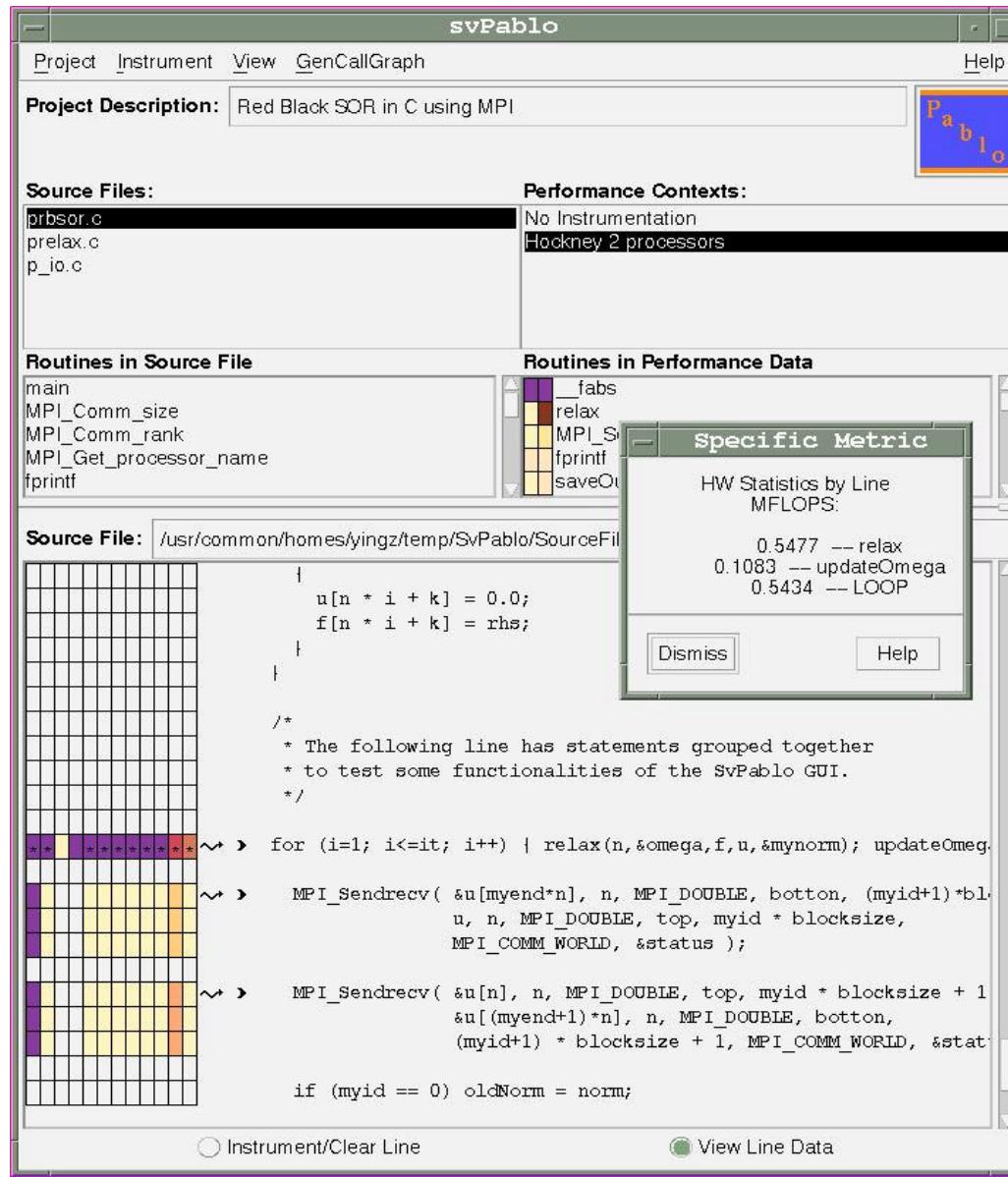
OK Help



Line Level Performance Data



Line Level Performance Data



Line Level Performance Data

Performance Data: Loop Statistics // LOOP

Tasks: 0 .. 3
File Name(s): prelax.c Routine Name: relax()
Line Number: 126

Source Code Fragment:

```
for ( i = mystart; i <= myend; i++ )  
{  
    for ( k = lsw; k < n-1; k += 2 )  
    {  
        resid = u[n*(i+1) + k] + u[n*(i-1) + k] +
```

Field Name	Mean	Max	Task	Min	Value	Task
Count	50.000000	50.000000	0	50.000000		
Seconds	203.263432	205.499691	1	200.909828		
Exclusive Seconds	97.797499	98.851190	0	96.634679		
FP Instructions	111442377.000000	111722098.000000	0	110604979.000000		
D1 Load Misses	6042052.250000	6132786.000000	0	5957914.000000		

View detailed performance data

Detailed Performance Data: HW Statistics by Line

Task Number	Count	Seconds	Exclusive Seconds	FP Instructions	D1 Load Misses	Branch Instructions	Branches
0	50	204.3373	98.8512	111722098.0000	6132786.0000	2882755555.0000	
1	50	205.4997	98.3902	111721239.0000	5957914.0000	2906441753.0000	

Dismiss Help

Source code segment

Statistical data

Per-task data



Experiments With POP Code

POP Overview

☞ **POP: Parallel Climate Model / Ocean Simulation, LANL**

☞ **Code Features**

- ☞ all source code in **F90**
- ☞ **MPI**-based communication
- ☞ more than 60 source files
- ☞ version analyzed: **POP-1.4.3**

☞ **Execution environment**

- ☞ IBM-SP3 at NERSC - Seaborg
- ☞ each node with 16 processors
- ☞ PAPI support enabled
- ☞ compiler used: xlf, -O3
- ☞ input data set: 1 degree case
- ☞ simulation limit: 20 time steps

Original POP Reporting

average # scans = 243

Timing information:

Time in timer: ANISO

Timer number 1 = 13.29 seconds

Time in timer: GM

Timer number 2 = 18.17 seconds

Time in timer: KPP

Timer number 3 = 13.18 seconds

Time in timer: IMPVMIXT

Timer number 4 = 3.31 seconds

Time in timer: IMPVMIXU

Timer number 5 = 1.42 seconds

Time in timer: TOTAL

Timer number 6 = 80.82 seconds

Time in timer: STEP

Timer number 7 = 80.82 seconds

Time in timer: BAROCLINIC

Timer number 8 = 69.51 seconds

Time in timer: BAROTROPIC

Timer number 9 = 6.01 seconds

}

← Major Subroutines

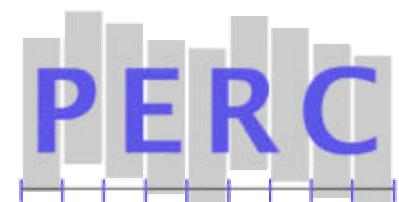
}

← Total Execution Time

← Major Code Regions

POP exiting...

Successful completion of POP run



POP Instrumentation

✍ Original POP Reporting

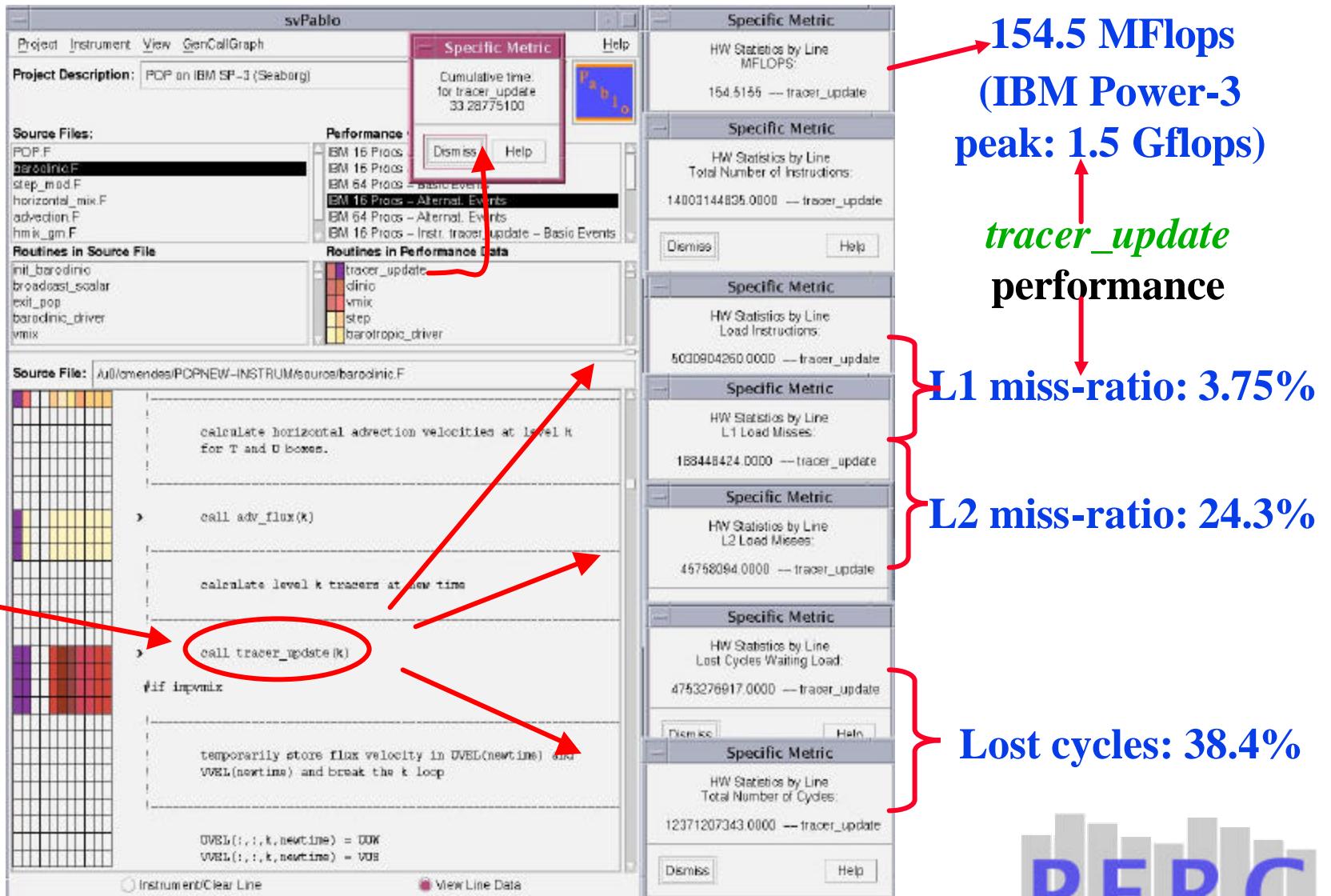
- ✍ Good indication of critical code regions
- ✍ Lack of quantitative performance data
- ✍ Lack of indication of major performance factors

✍ SvPablo Instrumentation Strategy

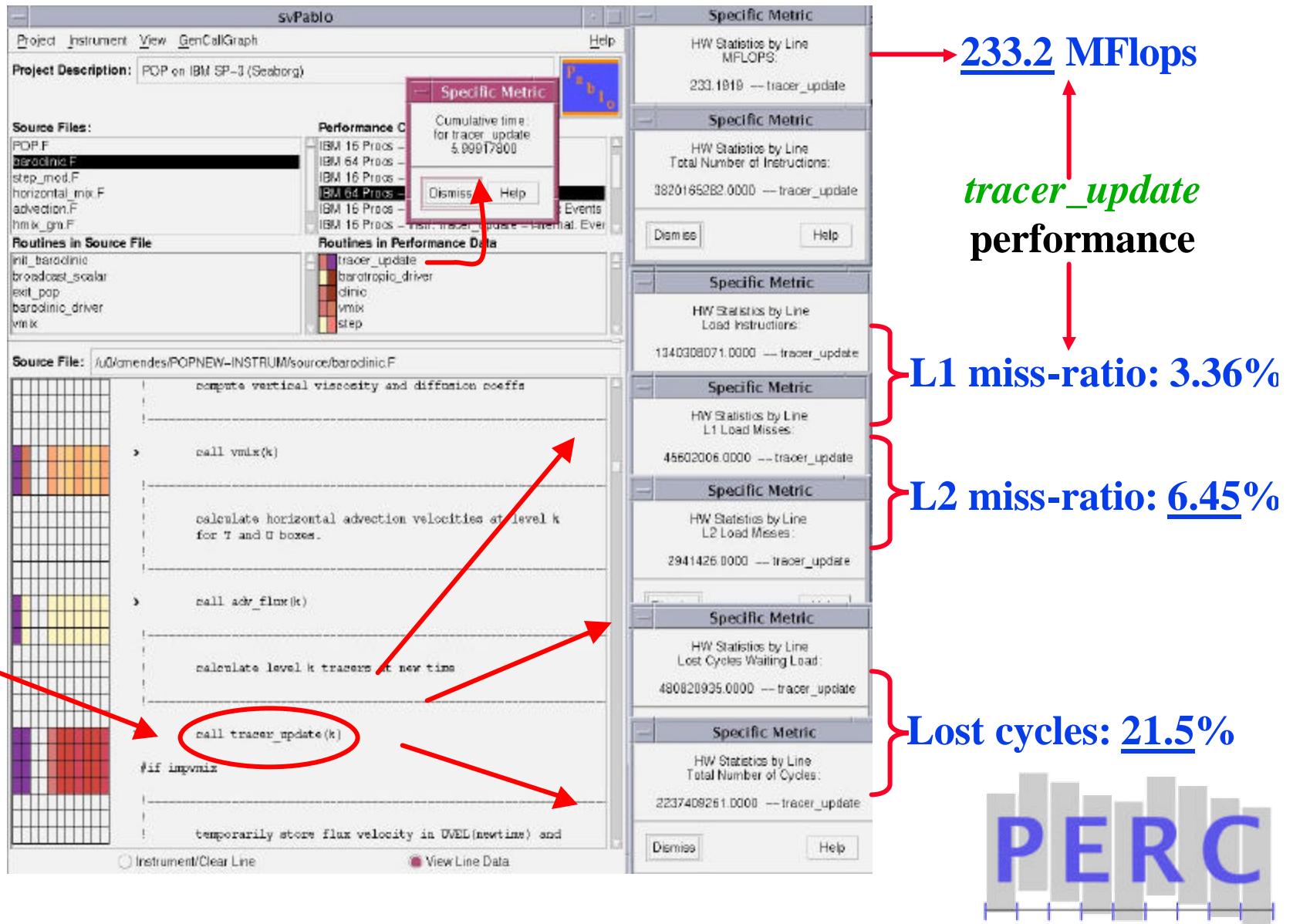
- ✍ Phase 1: higher level routines
 - ✍ verify consistency of original POP reporting
- ✍ Phase 2: lower level critical routines
 - ✍ capture detailed performance data
 - ✍ understand causes of observed performance
 - ✍ propose changes for improving performance

POP Performance on 16 Processors

Major routine in
baroclinic

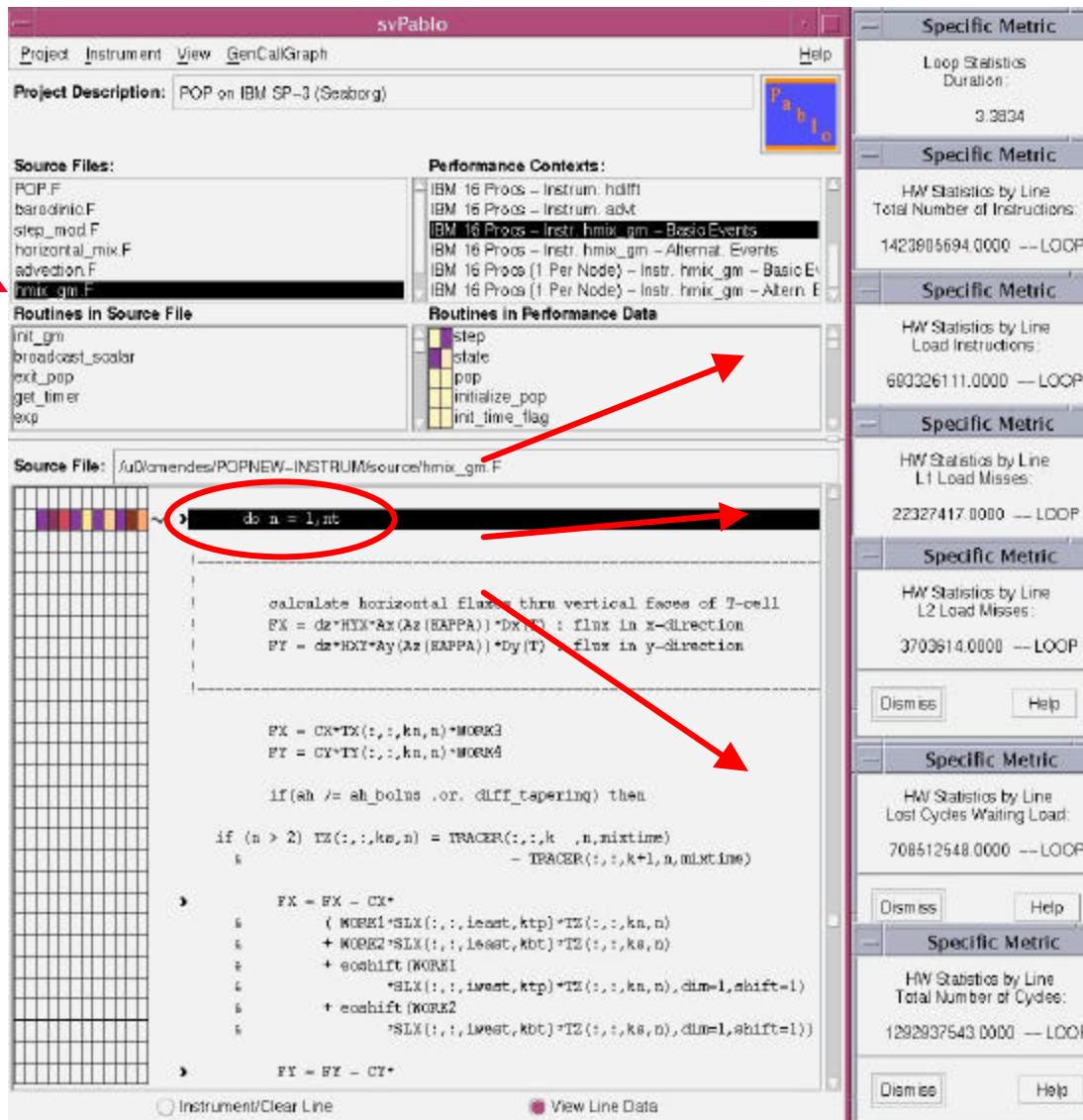


POP Performance on 64 Processors



POP Details on 16 Processors

Called by
tracer_update



loop
performance

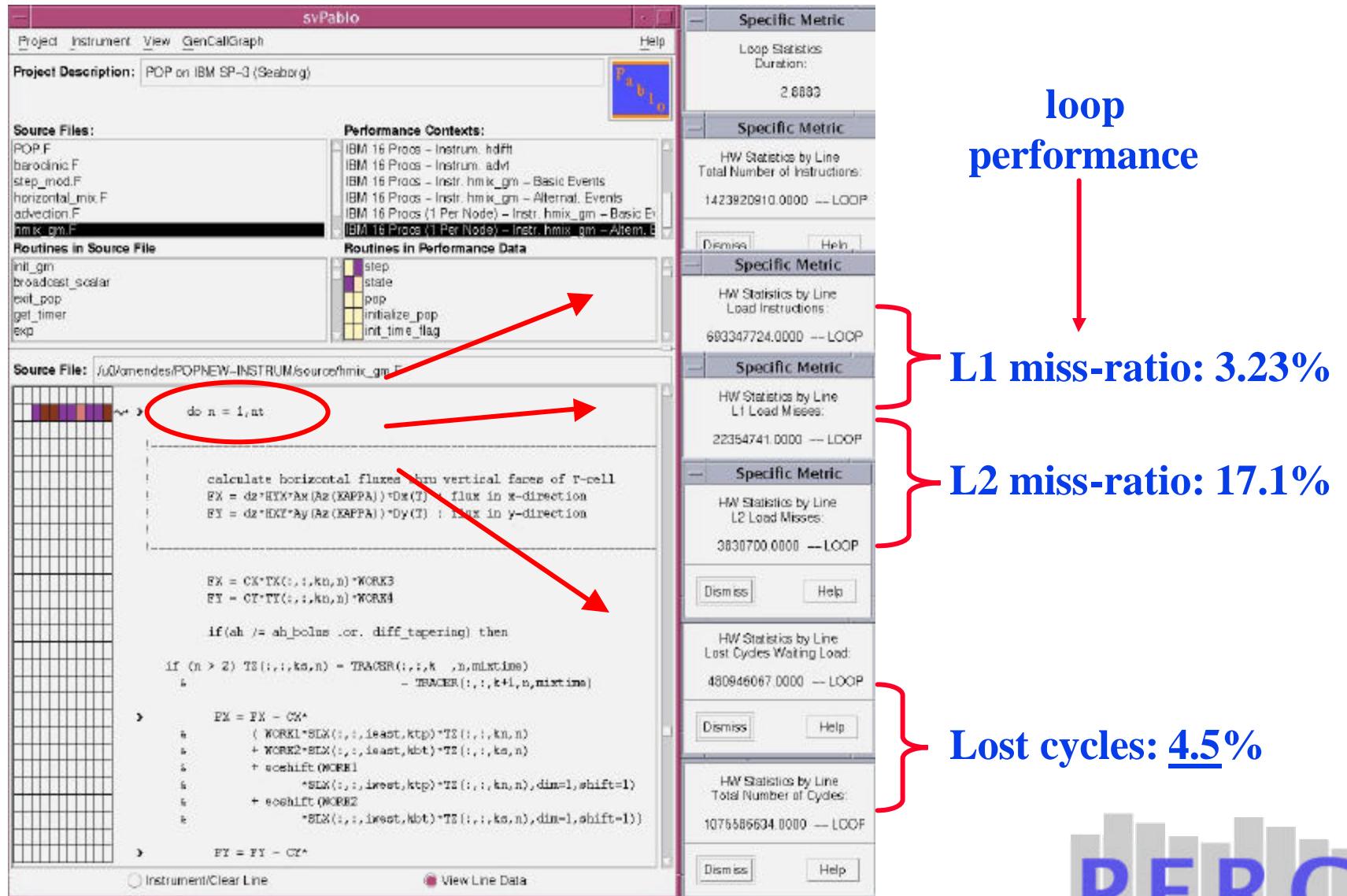
L1 miss-ratio: 3.22%

L2 miss-ratio: 16.6%

Lost cycles: 54.8%



POP on 16 Processors, 1 Per Node



POP: Memory Access Effects

✍ Observations

- ✍ Non-negligible L2 miss-ratios
- ✍ Main memory contention may be high

✍ Total POP execution times for P=16

- ✍ 1 node, 16 procs/node: 80.82 seconds
- ✍ 2 nodes, 8 procs/node: 78.45 seconds
- ✍ 4 nodes, 4 procs/node: 77.25 seconds
- ✍ 8 nodes, 2 procs/node: 77.31 seconds
- ✍ 16 nodes, 1 proc/node: 75.70 seconds

✍ Performance impact

- ✍ Measured improvements up to 6.4%
- ✍ Probably higher for larger problem cases

POP: Lessons Learned

✍ Conclusions

- ✍ memory access is a key issue on POP-1.4.3
- ✍ high L2 miss-ratios imply more main-memory traffic
- ✍ fat SMP nodes are bad for POP-1.4.3

✍ Performance improvement guidelines

- ✍ minimize memory sharing between processors
- ✍ improve L2 miss-ratio by code restructuring
- ✍ restructuring needs to be applied to major data structures (several code regions)

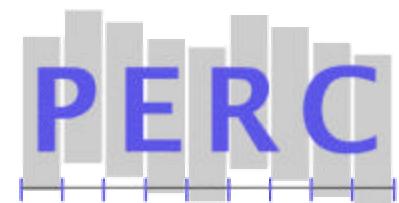
SvPablo Current Development

- ☞ **C++ code instrumentation and Performance data capture**
 - ☞ Integration with [ROSE](#), via SciDAC/PERC
- ☞ **OpenMP constructs instrumentation**
 - ☞ Limited support on Sun Solaris and IBM SP
- ☞ **Capture and display of communication data**
 - ☞ via MPI profiling interface
- ☞ **Scalability analysis**
- ☞ **Application signature modeling**
 - ☞ Signature modeling and comparison across platforms and across execution on the same platforms.
- ☞ **Instrumentation Overhead controlling**



Hands-on Session Overview

- ✍ **POP code instrumentation and performance data browsing demonstration**
 - ✍ Performance data is captured on Hockney on 8 nodes, 2 processors on each node
 - ✍ Two performance contexts
 - ✍ With different PAPI hardware counter event set
 - ✍ Two sets of performance data
 - ✍ [Sample Makefile](#)
 - ✍ [Sample instrumentation files](#)
- ✍ **Hands-on SvPablo practice with Red-Black SOR code**
 - ✍ A short C code with 3 source files
 - ✍ [Sample Makefile](#)
 - ✍ Test bed for practicing the whole cycle of using SvPablo



POP Code Demonstration

- ☞ Launch SvPablo GUI

```
% /usr/common/perc/scicomp8/SvPablo.demo/bin/runSvPablo
```

- ☞ Pull down “**Projects**” menu and “**Open Projects**”
- ☞ Select **POP** project
- ☞ Select performance context from “**Performance Contexts**” box
- ☞ Click on any one of the source code or routines in “**Routines in Performance Data**”
- ☞ Browse the source code and performance data
- ☞ **Left click** on the colored boxes will show various data value
- ☞ **Left click** on the instrumented line will show detailed function call or loop performance information
- ☞ **Left click** on any routines in “**Routines in Performance Data**” will show procedure level statistical performance data
- ☞ More display options available by pulling down “**View**” menu in the top menu bar



Hands-on Practice with Red-Black SOR

- ☞ Load SvPablo module into home directory

```
% module load use.perc  
% module load scicomp8/SvPablo.module
```

- ☞ Instrument source code

- ☞ Launch SvPablo GUI

```
% cd $(home)/SvPablo.module/bin  
% ./runSvPablo
```

- ☞ Open project CMPI_RBSOR

- ☞ Select “Example: hockney 2 processors” context to see the sample instrumentation and performance data

- ☞ Select “User Experiments” context to start your own instrumentation

- ☞ Save your instrumentation by pulling down “Instrument” menu and select “Save Instrumentation”



Hands-on Practice with Red-Black SOR

✍ Compile and run the instrumented code

✍ Go to source directory

```
$(home)/SvPablo.module/SourceFiles/CMPI_RBSOR
```

✍ Modify **Makefile.inst** as instructed in README in this directory

- ✍ Set variable **\$(CONTEXT)** to **UserInst**

- ✍ Set **\$(NP)** to the number of processors desired

- ✍ Reset source file names to instrumented file names

✍ Compile the instrumented code

```
% gmake -f Makefile.Inst
```

- ✍ Will automatically compile, run the executable, and generate performance file

- ✍ The performance file name is by default named as **svPabloPerformanceFile.xx**, where xx is process ID.

✍ Copy the performance file into performance context directory

```
% cp svPabloPerformanceFile.xx
```

```
$(home)/SvPablo.module/svPabloProjects/CMPI_RBSOR/UserInst
```



Hands-on Practice with Red-Black SOR

- ☞ **Browsing the performance data**
 - ☞ In GUI, pull down “**Projects**” menu and select “**Edit Projects**”
 - ☞ In “**Edit Project**” box, click on “**UserInst**” context -> “**Change**” to launch context editing box
 - ☞ In “**Edit Performance Context**” box, load the performance file by clicking on “**Change**” button.
 - ☞ Click “**OK**” on all boxes
- ☞ **In GUI, click on “**User Experiments**” context**
- ☞ **Begin to browse your performance data**

Hey, you did it!

